
Lecture4(part2)

Topics covered:
Arithmetic



Multiplication of signed-operands

- ❑ Recall we discussed multiplication of unsigned numbers:
 - ◆ Combinatorial array multiplier.
 - ◆ Sequential multiplier.
- ❑ Need an approach that works uniformly with unsigned and signed (positive and negative 2's complement) n -bit operands.
- ❑ Booth's algorithm treats positive and negative 2's complement operands uniformly.



Booth's algorithm

- ❑ Booth's algorithm applies uniformly to both unsigned and 2's complement signed integers.
 - ◆ Basis of other fast product algorithms.
- ❑ Fundamental observation:
 - ◆ Division of an integer into the sum of block-1's integers.

Suppose we have a 16-bit binary number: 0110011011110110

This number can be represented as the sum of 4 “block-1” integers:

```
0110000000000000
0000011000000000
0000000011110000
+0000000000000110
0110011011110110
```

◆ Booth's algorithm (contd..)

Suppose Q is a block-1 integer: $Q = 0000000001111000 = 120$

Then: $X.Q = X.120$

Now: $120 = 128 - 8$, so that $X.Q = X.120 = X.(128-8) = X.128 - X.8$

And:

$Q = 000000000\textcolor{red}{0}111\textcolor{blue}{1}000$

$128 = 000000000\textcolor{red}{1}0000000$

$8 = 0000000000000\textcolor{blue}{1}000$

If we label the LSB as 0, then the first 1 in the block of 1's is at position 3 and the last one in the block of 1's is at position 6.

As a result:

$$X.Q = X.120 = X.128 - X.8 = X.2^7 - X.2^3$$



Booth's algorithm (contd..)

Representing Block-1 integers

Q is an n -bit block-1 unsigned integer:

- Bit position 0 is LSB.
- First 1 is in bit position j
- Last 1 is in bit position k

Then:

$$Q = 2^{k+1} - 2^j$$

$$Q.X = X.(2^{k+1} - 2^j) = X.2^{k+1} - X.2^j$$

◆ Booth's algorithm (contd..)

Let Q be the block-1 integer: $Q = 0111111111111110$

To form the product $X.Q$ using normal multiplication would involve 14 add/shifts (one for each 1-valued bit in multiplier Q).

Since:

$$Q = 2^{15} - 2^1$$
$$X.Q = X.(2^{15} - 2^1)$$

Product $X.Q$ can be computed as follows:

1. Set the Partial Product (PP) to 0.
2. Subtract $X.2^1$ from PP.
3. Add $X.2^{15}$ to PP.

Note that $X.2^j$ is equivalent to shifting X left j times.

◆ Booth's algorithm (contd..)

If Q is not a block-1 integer, Q can be decomposed so that it can be represented as a sum of block-1 integers.

Suppose: $Q = 0110011011110110$

Q can be decomposed as:

$$\begin{array}{r} 0110000000000000 = 2^{15} - 2^{13} \\ 0000011000000000 = 2^{11} - 2^9 \\ 0000000011110000 = 2^7 - 2^4 \\ + \underline{0000000000000110} = 2^3 - 2^1 \\ 0110011011110110 \end{array}$$

Thus,

$$Q.X = X.(2^{15} - 2^{13} + 2^{11} - 2^9 + 2^7 - 2^4 + 2^3 - 2^1)$$



Booth's algorithm (contd..)

Inputs: n -bit multiplier Q
 n -bit multiplicand x
 $2n$ -bit current Partial Product (PP) initially set to 0.
 (Upper half of PP is bits $n-1$ through n)
 Q has an added '0' bit attached to the LSB (Q has $n+1$ bits).

Algorithm: For every bit in Q :

1. Examine the bit and its neighbor to the immediate right.
 If the bit pair is:
 - 00 – do nothing.
 - 01 – Add the multiplicand to the upper half of PP.
 - 10 – Sub the multiplicand from the upper half of PP.
 - 11 – Do nothing.
2. Shift the PP right by one bit, extending the sign bit.

Signed multiplication

Sign extension is shown in blue

Figure 6.8 Sign extension of negative multiplicand.



Booth Multiplier Encoding

Multiplier		Version of multiplicand selected by bit i
Bit i	Bit $i - 1$	
0	0	$0 \times M$
0	1	$+1 \times M$
1	0	$-1 \times M$
1	1	$0 \times M$

Figure 6.12 Booth multiplier recoding table.


0	0	1	0	1	1	0	0	1	1	1	0	1	0	1	1	0	0
																	
0	+1	-1	+1	0	-1	0	+1	0	0	-1	+1	-1	+1	0	-1	0	0

Figure 6.10 Booth recoding of a multiplier.



Booth Multiplication with negative multiplier

$$\begin{array}{r} 01101 \ (+13) \\ \times 11010 \ (-6) \\ \hline \end{array} \quad \Rightarrow \quad \begin{array}{r} 01101 \\ 0-1+1-1\ 0 \\ \hline \end{array}$$
$$\begin{array}{r} 0000000000 \\ 111110011 \\ 00001101 \\ 1110011 \\ 000000 \\ \hline 1110110010 \ (-78) \end{array}$$

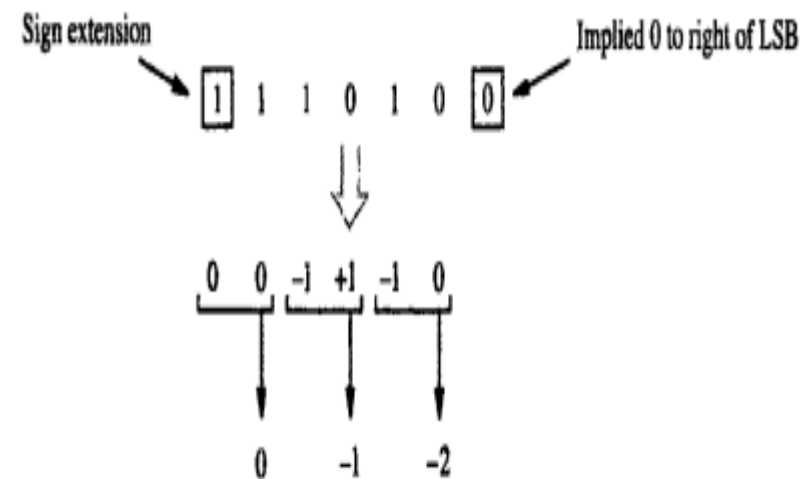
Fast Multiplication

Bit-pair encoding of multiplier

Multiplier bit-pair		Multiplier bit on the right $i-1$	Multiplicand selected at position i
$i+1$	i		
0	0	0	$0 \times M$
0	0	1	$+1 \times M$
0	1	0	$+1 \times M$
0	1	1	$+2 \times M$
1	0	0	$-2 \times M$
1	0	1	$-1 \times M$
1	1	0	$-1 \times M$
1	1	1	$0 \times M$

(b) Table of multiplicand selection decisions

Figure 6.14 Multiplier bit-pair recoding.





Booth encoding versus Bit-pair encoding of multiplier

$$\begin{array}{r} 01101 \text{ (+13)} \\ \times 11010 \text{ (-6)} \\ \hline \end{array}$$



$$\begin{array}{r} 01101 \\ 0-1+1-10 \\ \hline 00000000000 \\ 111110011 \\ 00001101 \\ 1110011 \\ 000000 \\ \hline 1110110010 \text{ (-78)} \end{array}$$

Using Booth encoding

$$\begin{array}{r} 01101 \\ 0-1-2 \\ \hline 1111100110 \\ 11110011 \\ 000000 \\ \hline 1110110010 \end{array}$$

Using bit-pair encoding

◆ Unsigned division

- Division is a more tedious process than multiplication.
- For the unsigned case, there are two standard approaches:
1.) Restoring division. 2.) Non restoring division.

$$\begin{array}{r} 21 \\ 13 \overline{) 274} \\ \underline{26} \\ 14 \\ \underline{13} \\ 1 \end{array}$$

Try dividing 13 into 2.

Try dividing 13 into 26.

$$\begin{array}{r} 10101 \\ 1101 \overline{) 10001001} \\ \underline{01101} \\ 10000 \\ \underline{1101} \\ 1110 \\ \underline{1101} \\ 1 \end{array}$$

Try dividing 1101 into 1, 10, 100, 1000 and 10001.

◆ Restoring division

How do we know when the divisor has gone into part of the dividend correctly?

$$\begin{array}{r} 10101 \\ 1101 \overline{) 10001001} \\ \underline{01101} \\ 10000 \\ \underline{1101} \\ 1110 \\ \underline{1101} \\ 1 \end{array}$$

Subtract 1101 from 1, result is negative
Subtract 1101 from 10, result is negative.
Subtract 1101 from 100, result is negative
Subtract 1101 from 1000, result is negative.
Subtract 1101 from 10001, result is positive.



Restoring division

Strategy for unsigned division:

Shift the dividend one bit at a time starting from MSB into a register.
Subtract the divisor from this register.

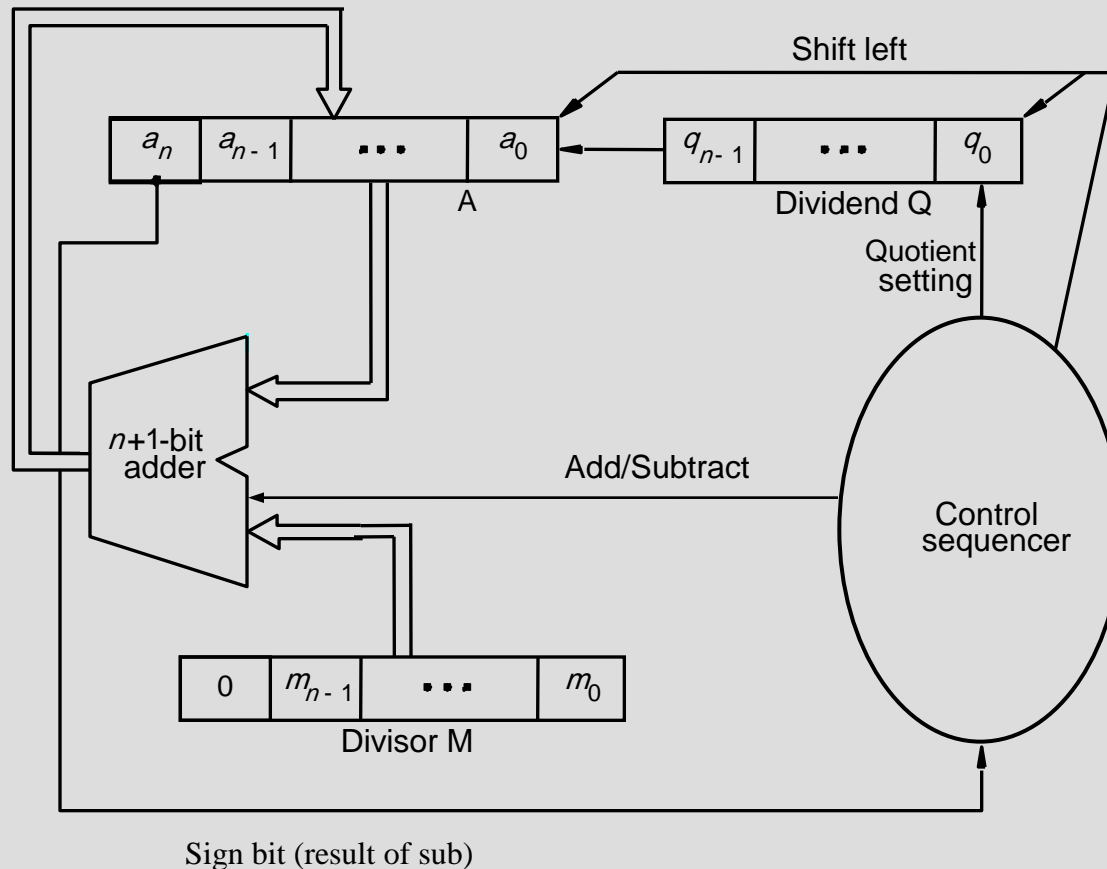
If the result is negative ("didn't go"):

- Add the divisor back into the register.
- Record 0 into the result register.

If the result is positive:

- Do not restore the intermediate result.
- Set a 1 into the result register.

◆ Restoring division (contd..)



Set Register A to 0.

Load dividend in Q.

Load divisor into M.

Repeat n times:

- Shift A and Q left one bit.*
- Subtract M from A.*
- Place the result in A.*
- If sign of A is 1, set q_0 to 0 and add M back to A.*
- Else set q_0 to 1.*

End of the process:

- Quotient will be in Q.*
- Remainder will be in A.*

Restoring division (contd..)

$$\begin{array}{r}
 10 \\
 11 \overline{) 1000} \\
 \underline{11} \\
 10
 \end{array}$$

Initially	0 0 0 0 0	1 0 0 0	} First cycle
Shift	0 0 0 0 1	0 0 0 	
Subtract	<u>1 1 1 0 1</u>		
Set q_0	1 1 1 1 0		
Restore	<u>1 1</u>		} Second cycle
	0 0 0 0 1	0 0 0 0	
Shift	0 0 0 1 0	0 0 0 	
Subtract	<u>1 1 1 0 1</u>		
Set q_0	1 1 1 1 1		} Third cycle
Restore	<u>1 1</u>		
	0 0 0 1 0	0 0 0 0	
Shift	0 0 1 0 0	0 0 0 	
Subtract	<u>1 1 1 0 1</u>		} Fourth cycle
Set q_0	0 0 0 0 1		
Shift	0 0 0 1 0	0 0 0 1	
Subtract	<u>1 1 1 0 1</u>	0 0 1 	
Set q_0	1 1 1 1 1		}
Restore	<u>1 1</u>		
	0 0 0 1 0	0 0 1 0	
	<u>Remainder</u>	<u>Quotient</u>	

Non-restoring division

Restoring division can be improved using non-restoring algorithm

The effect of restoring algorithm actually is:

If A is positive, we shift it left and subtract M , that is compute $2A - M$

If A is negative, we restore it ($A + M$), shift it left, and subtract M , that is, $2(A + M) - M = 2A + M$.

Set q_0 to 1 or 0 appropriately.

Non-restoring algorithm is:

Set A to 0.

Repeat n times:

If the sign of A is positive:

Shift A and Q left and subtract M . Set q_0 to 1.

Else if the sign of A is negative:

Shift A and Q left and add M . Set q_0 to 0.

If the sign of A is 1, add A to M .

Non-restoring division (contd..)

$$\begin{array}{r} 10 \\ 11 \overline{) 1000} \\ \underline{11} \\ 10 \end{array}$$

The diagram illustrates the steps of the restoring division algorithm for 5-bit numbers. The process involves four cycles of shifting and subtracting, with the quotient and remainder being restored at the end.

Initial Setup:

- Initially: Divisor (00001), Dividend (10000), q_0 (0000)

First cycle:

- Shift: Divisor (00001), Dividend (00001), q_0 (0000)
- Subtract: Divisor (11101), Dividend (10000), q_0 (1110)
- Set q_0 : The 5th bit of q_0 is set to 0 (00000).

Second cycle:

- Shift: Divisor (11100), Dividend (00000), q_0 (0000)
- Add: Divisor (00011), Dividend (00000), q_0 (1111)
- Set q_0 : The 5th bit of q_0 is set to 0 (0000).

Third cycle:

- Shift: Divisor (11100), Dividend (00000), q_0 (0000)
- Add: Divisor (00011), Dividend (00000), q_0 (1111)
- Set q_0 : The 5th bit of q_0 is set to 0 (0000).

Fourth cycle:

- Shift: Divisor (00010), Dividend (00010), q_0 (0001)
- Subtract: Divisor (11101), Dividend (00010), q_0 (0001)
- Set q_0 : The 5th bit of q_0 is set to 0 (0000).

Final Result:

- Quotient: 1111
- Remainder: 0001

The diagram shows the restoration of the remainder by adding the divisor (11101) to the current remainder (00010), resulting in the final remainder (00010).